Extension of TAPENADE towards FORTRAN9X

Valérie Pascual, Laurent Hascoët
TROPICS team, INRIA Sophia-Antipolis, France

1 Introduction

We present the extensions to the Automatic Differentiation tool TAPENADE [5] towards FORTRAN9X [7]. Other AD tools already took this direction, such as TAF [3, 6]. ADIFOR [2] already considered the question of structured types.

We recall the internal architecture of TAPENADE, with a central module for program analysis and transformation, surrounded by language-specific front-ends and back-ends. This allows the central module to forget about mostly syntactic details of the analyzed language, and to concentrate on the language's semantic constructs. To this end, TAPENADE defines an internal abstract language, called IL, able to represent all constructs of classical imperative languages. In particular, extension to FORTRAN9X drove us to add several new constructors into IL. Furthermore, programs are internally represented as Call Graphs, Control Flow Graphs [1], and syntax trees only at the deepest level of individual instructions. This yields a general representation for all control structures.

Concerning FORTRAN9X concrete syntax, everything is taken care of by a specific new parser (front-end) and pretty-printer (back-end). The main novelty with respect to FORTRAN77 is the long awaited "free format", where statements may start in column 1 instead of 7. Our new FORTRAN9X parser accepts programs that combine the old "fixed format" and the free format, and the back-end can regenerate programs using both formats. Thanks to the internal representation as Control Flow Graphs, new constructs such as the `SELECT CASE` were easily added and treated by the differentiation engine as any other flow of control.

In this paper, we focus on the new features of FORTRAN9X that required us to make important choices and improvements into TAPENADE. We put these features into four categories: section 2 deals with the nesting of modules, subprograms, and interfaces, section 3 deals with the introduction of structured types, unfortunately called "derived" types, section 4 deals with the new overloading capabilities, and section 5 deals with array notation for vectorial computers. Section 6 concludes on the soon to come pointer analysis, and the more distant extension to objects. The full version of this paper presents some illustrative examples.

2 Nesting of modules and subprograms

The internal representation had to be extended to capture the new nesting capabilities, with modules, internal/external subprograms, and interfaces. The structure of FORTRAN77 was flat, apart from internal subprograms in some dialects.

We chose to introduce an internal tree representation of modules nesting, in addition to the existing Call Graph. Each node stands for one "unit", i.e. subprogram or module, and holds the list of its enclosed units. In particular, the regenerated program must comply with this unit tree structure, so that this program is stand-alone and can be compiled directly. Each unit defines two symbol tables, for the public and private symbols (i.e. arguments, variables, subprograms, types, etc). Of course private symbol table is nested into the public. Symbol table nesting already existed in TAPENADE for scoping. The `USE` statement just states that a unit has access to the public symbol table of a module.

Classically, program analyses and transformations need to run in an appropriate order on the subprograms, depending on the Call Graph. The novelty is that this order now must take into account the `USE` of modules. Moreover, recursion may introduce cycles in this dependence, so the best order can only be an approximation.

When differentiation is concerned, the question is what must belong to a differentiated unit. When FORTRAN77 was concerned, differentiated symbols could be defined independently from their original symbols. Now that modules can define their own private symbols, some differentiated unit must often be declared in the same context as its original unit, i.e. must live inside the same enclosing module. In general the question is whether the differentiated object can exist independently of the original object, or must they be attached inside the same enclosing level. For example a differentiated instruction must be in the same subprogram as the original instruction because they share a common control. Similarly a differentiated subprogram must be in the same module as the original if both access a private field of this module. On the contrary, a differentiated field `x` of a structured type `T` need not be added into `T`, but can rather go into a stand-alone "differentiated" structured type `T'`, therefore saving memory space.

3 Structured or "derived" types

FORTRAN9X allows the user to define "derived" types in order to manipulate composite objects containing several components. As we said above,

our choice during differentiation is to define a differentiated structured type, whose fields hold the derivatives of the original fields. However, it happens that some variables of a structured type have only some components that are active. Then the differentiated type need not allocate space for the other components. Differentiated structured types may depend on the activity pattern. On the other hand, we don't want to *specialize* too far, creating several differentiated types for a given structured type. Therefore, our choice is very similar to differentiation of subprograms with several activity patterns: there is only one differentiated type `T'` for each structured type `T`. During activity analysis, if a component `x` of some variable of type `T` can be active, the differentiated type `T'` must hold a component `x` too. The price for this non-specialization is that sometimes, a differentiated variable of type `T'` will not use all of its components.

4 Overloading

Overloading is being able to call different subprograms by the same generic name. Whereas overloading in Object Oriented languages is resolved only at run time, the limited form of overloading in FORTRAN9X can be resolved statically at compile time, and therefore at differentiation time. This is done during the type-checking phase. Notice furthermore that FORTRAN9X also allows the user to overload predefined operators such as `+`, `-`, `*`, `/`, or the assignment `=`.

We must thus modify the type-checking algorithm carefully. Each use of a predefined operator or call to a subprogram is compared to available overloaded subprograms according to the arguments' types. If necessary, it is replaced by a standard subprogram call, and treated as such in the following differentiation phase. After type-checking stage, overloading is resolved.

When differentiation is concerned, the predefined operators are treated in a very particular, built-in manner. So we must be careful not to replace these operators by ordinary subprograms calls when not necessary.

5 Array features

The vectorial programming concepts of FORTRAN9X are represented through syntactic notations and intrinsic functions. In FORTRAN9X programs, it is possible to use arrays globally. In the differentiated program, we keep this property whenever possible and we also treat differentiated objects globally.

Notice that array notation can be advantageous for static data flow anal-

ysis. A global reset of an array to a constant, for example, can easily be detected and the array is considered "killed", whereas the equivalent loop requires array region analysis to reach the same conclusion.

When differentiation is concerned, intrinsic array functions are divided in two categories: the `SUM` intrinsic and the `SPREAD` intrinsic (which is often implicit) have a special behavior. In particular the adjoint of a `SUM` is a `SPREAD`, and vice-versa. Actually these two intrinsics blend into the internal representation for partial derivatives, and reappear when generating the differentiated code.

For example the following vectorial instruction:
`A(0:100:3) = x * SUM(B(:))`
is differentiated in the reverse mode using the transposed local Jacobian:

$$\left\{\frac{\partial[A,x,B]_{out}}{\partial[A,x,B]_{in}}\right\}^* = \begin{pmatrix} 0 & \texttt{SUM(B(:))} & \texttt{x} \\ 0 & Id & 0 \\ 0 & 0 & Id \end{pmatrix}^* = \begin{pmatrix} 0 & 0 & 0 \\ \texttt{SUM(B(:))} & Id & 0 \\ \texttt{x} & 0 & Id \end{pmatrix}$$

which is a block matrix and yields the adjoint instructions:
`x̄ = x̄ + SUM(B(:))*SUM(Ā(0:100:3))`
`B̄(:)= B̄(:)+ x*SUM(Ā(0:100:3))`
`Ā(0:100:3) = 0.0`

All the other array intrinsics are treated like black-box routines, whose differentiation is given to TAPENADE in special library files.

6 Conclusion

The next development of TAPENADE for FORTRAN9X will concern pointers and dynamic allocation. We plan to share this with the programmed extension to C. In the long run, we also consider extending TAPENADE to the object programming concepts from C++ or JAVA. This will introduce dynamic overloading, which is still a challenge for automatic differentiation. For example, the activity pattern of the actual parameters of a given call may contribute to the activity pattern of several subprograms.

TAPENADE can be utilized as a server at the url
`http://tapenade.inria.fr:8080/tapenade/index.jsp`
It can be downloaded from `ftp://ftp-sop.inria.fr/tropics/tapenade`

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] A. Carle and M. Fagan. ADIFOR 3.0 overview. Technical Report CAAM-TR-00-02, Rice University, 2000.

[3] R. Giering, T. Kaminski, and T. Slawig. Applying TAF to a Navier-Stokes solver that simulates an Euler flow around an airfoil. In *To appear in Future Generation Computer Systems*. Elsevier Science, 2004. [http://www.fastopt.com/papers/gieringal02.pdf].

[4] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, Frontiers in Applied Mathematics, 2000.

[5] INRIA Tropics team. On-line documentation of the TAPENADE AD tool. Technical report. [http://www.inria.fr/tropics].

[6] T. Kaminski, R. Giering, M. Scholze, P. Rayner, and W. Knorr. An example of an automatic differentiation-based modelling system. In *Computational Science - ICCSA 2003 - Lecture Notes in Computer Science*. Springer, 2003. [http://www.fastopt.com/papers/kaminskial03.pdf].

[7] M. Metcalf and J. Reid. *FORTRAN 90/95 explained*. Oxford University Press, 1996.